R Small Group: Class 1

Amy Allen & Sara Snell

June 14, 2017

Using this document

- Code blocks and R code have a grey background (note, code nested in the text is not highlighted in the pdf version of this document but is a different font).
- # indicates a comment, and anything after a comment will not be evaluated in R
- The comments beginning with ## under the code in the grey code boxes are the output from the code directly above; any comments added by us will start with a single #
- While you can copy and paste code into R, you will learn faster if you type out the commands yourself.
- Read through the document after class. This is meant to be a reference, and ideally, you should be able to understand every line of code. If there is something you do not understand please email us with questions or ask in the following class (you're probably not the only one with the same question!).

Class 1 expecations

1. Know and understand the basic math operators (`+`, `-`, `*`, `/`, `^`)
2. Know the assignment operator and how to use it
3. Understand what a function is, how to use a function, and understand some basic functions
4. Understand the three most common data classes (character, numeric, logical)
5. Know and understand the basic comparison operators (`>`, `<`, `==`, `>=`, `<=`)
6. Understand how to compare objects, and predict the data classes and how they change when comparing or combining objects

Introduction to R

The most basic introduction to R is through simple math calculations. For example:

```
3 + 4
## [1] 7
```

Similarly, R can do subtraction, multiplication, division, and exponentiation:

```
4 - 2
## [1] 2
2*120
## [1] 240
1e3/2
## [1] 500
3^2
## [1] 9
```

The `<-` command is used to store variables in R. For example, if we want to store the result of `3 + 4` to use in future calculations, we could do something like the following:

```
x <- 3 + 4
```

Notice, code did not return any result. The result is now stored in `x`. In the example above, “x” is the name of the variable. The `<-` operator told R to calculate `3 + 4` and store the results in a variable called `x`. You can now access the variable by typing it into the console:

```
x
## [1] 7
```

The `x` variable can also be used in calculations and to create new variables.

```
y <- 2 + x
y
## [1] 9
```

It can be hard to remember which variables are available. R provides the `ls` function for displaying the variables contained in your environment. Functions have names, in this case `ls`. What happens when you run `ls`?

```
ls
## function (name, pos = -1L, envir = as.environment(pos), all.names = FALSE,
##     pattern, sorted = TRUE)
## {
##     if (!missing(name)) {
##         pos <- tryCatch(name, error = function(e) e)
##         if (inherits(pos, "error")) {
##             name <- substitute(name)
##             if (!is.character(name))
##                 name <- deparse(name)
##             warning(gettextf("%s converted to character string",
##                 sQuote(name)), domain = NA)
##             pos <- name
##         }
##     }
##     all.names <- .Internal(ls(envir, all.names, sorted))
##     if (!missing(pattern)) {
##         if ((ll <- length(grep("[", pattern, fixed = TRUE))) &&
##             ll != length(grep("]", pattern, fixed = TRUE))) {
##             if (pattern == "[") {
##                 pattern <- "\\["
##                 warning("replaced regular expression pattern '[' by  '\\\\['")
##             }
##             else if (length(grep("[^\\\\]\\[<-", pattern))) {
##                 pattern <- sub("\\[<-", "\\\\\\[<-", pattern)
##                 warning("replaced '[<-' by '\\\\[<-' in regular expression pattern")
##             }
##         }
##         grep(pattern, all.names, value = TRUE)
##     }
##     else all.names
## }
## <bytecode: 0x7fc5e3e2b240>
## <environment: namespace:base>
```

The output shows the code behind the `ls` function. To use a function you have to add `()`.

```
ls()
## [1] "x" "y"
```

Running `ls()` returns `[1] "x" "y"`, indicating there are two variables, `x` and `y`, in the global environment. Here we see a new type of result: the `x` and `y` are surrounded by quotation marks. The quotation marks indicate a different type of data. Programming languages include different data classes. While R includes many different types, we will discuss the three most common: (1) numeric, (2) character, and (3) logical. We can check the data class with the `class` function.

```
vars <- ls()
class(x = vars)
## [1] "character"
class(x = x)
## [1] "numeric"
```

Notice the `class` function requires an input, in this case called "x". Running `class()` without any input will give an error message:

```
class()
## Error in class(): 0 arguments passed to 'class' which requires 1
```

In some programming languages the user has to specify the data class beforehand. R makes an educated guess on the data class, which is convenient, but can cause unexpected problems if the user is not careful. Let's explore data classes further after making some new variables.

```
var1 <- c(1, 2, 3)
var2 <- c("1", "2", "3")
var3 <- c("a", "b", "c")
```

The `c` function creates a vector – or a string of values. We did not tell R what data class to make the vectors. Run `class` to see how R interpreted the vectors.

```
class(var1)
## [1] "numeric"
class(var2)
## [1] "character"
class(var3)
## [1] "character"
```

Even though `var2` could be interpreted as numeric, adding the quotation marks created a character vector. When combining or comparing two values/vectors R attempts to coerce one to the higher order data class. From the R documentation:

> If the two arguments are atomic vectors of different types, one is coerced to the type of the other, the (decreasing) order of precedence being character, complex, numeric, integer, logical and raw.

Although we won't discuss all of the data classes listed above, it provides a nice reference for future use. (To find help in the documentation play around with `?` and `??`. `?` will show the documentation for a specific function, e.g. `?mean`, and `??` searches the documentation for a topic, e.g. `??average`. You can see from the

??average search that a quick Google search will often provide better results when you don't know the name of the function you are looking for.)

In addition to the algebraic operators discussed at the beginning of the class, R includes the comparison operators. We will illustrate how R interprets data classes and introduce the logical class using the comparison operators. < & > behave as you might expect:

```
1 < 2
## [1] TRUE
1 > 2
## [1] FALSE
```

Here, the comparison is simple because the values on both sides of the operator are numeric. (You can check the class, by running class on them, e.g. class(1) & class(2).) Notice the TRUE and FALSE outputs above. TRUE and FALSE are protected reserved terms in R, and cannot be modified. (If you are curious, try running FALSE <- TRUE. There are other reserved terms that can be overridden in the global environment. For example, LETTERS is a vector of capital letters, but can be overwritten by something like LETTERS <- "a".)

TRUE and FALSE are in the "logical" data class. We will discuss how they are used more when we talk about if/else statements later in the class, but they simply mean "true" and "false." So we would expect that 1 = 1 would return TRUE:

```
1 = 1
## Error in 1 = 1: invalid (do_set) left-hand side to assignment
```

The = operator is another assignment operator in R. We won't discuss the differences between <- and =, but know that you should almost always be using <-. We see here that numbers are also protected reserved terms in R. The correct operator for comparing if two items are equal in R is == – similarly greater than and less than equal operators are >= and <=.

```
1 == 1
## [1] TRUE
1 >= 1
## [1] TRUE
1 >= -1
## [1] TRUE
"1" == 1
## [1] TRUE
```

The last expression in the code above gives an interesting result. R interpreted "1" as equal to 1. We know from var2 above that adding quotes to a number indicates a character class rather than a numeric. But we also know from the documentation quote above that when two arguments are of different class, R will coerce the arguments to the data type with the highest precedence. In the expression above the 1 on the right side of the operator was coerced to the character data class, and R told you that "1" in fact equals "1". Similarly, if we combine var1 and var2 R will coerce var1 to a character.

```
class(c(var1, var2))
## [1] "character"
```

Don't let the code above intimidate you. The code just has two nested function calls. Look for the innermost parentheses and work outward. First, the expression called c to create a vector, made up of var1 and var2. Then the resulting vector was passed to class. If this is confusing, come back and run the two steps separately (hint: after you run the inner function and see that the numbers have quotations, you can store that resulting vector as a new variable, then check the data class of the new variable with `class`).

To make checking and manipulating data classes R provides the `as` and `is` functions. `is` will check whether the input is a specific data class and `as` will coerce the input to the given data class. Let's use these functions to explore how `TRUE` and `FALSE` are converted to other data classes.

```
is(object = TRUE, class2 = "logical")
## [1] TRUE
as(object = TRUE, Class = "numeric")
## [1] 1
as(object = FALSE, Class = "numeric")
## [1] 0
```

Notice that functions can have more than one input (parameter), and each input is named. Inspecting the `class` function closer would show that the name of the input parameter is "x". (If you do not give the name of the parameter R will assume the inputs were given in the order the parameters are defined for the function. Here, typing `is(TRUE, "logical")` would give the same output, but `is("logical", TRUE)` does not. However, `is(class2 = "logical", object = TRUE)` will give the same result because the parameter order does not matter if the parameters are specified.) Additionally, R provides shortcut functions for the different data classes.

```
is.character(TRUE)
## [1] FALSE
is.logical(FALSE)
## [1] TRUE
as.character(TRUE)
## [1] "TRUE"
```

Notice that coercing `TRUE` to a character and to a numeric provide different results. In numeric terms `TRUE` is `1` and `FALSE` is `0`. What about converting from a higher precedence class to a lower precedence class?

```
as.numeric("1")
## [1] 1
as.numeric("a")
## Warning: NAs introduced by coercion
## [1] NA
as.logical("a")
## [1] NA
as.logical("1")
## [1] NA
as.logical(as.numeric("1"))
## [1] TRUE
as.logical(1)
## [1] TRUE
as.logical(-10)
## [1] TRUE
as.logical(0)
## [1] FALSE
```

Notice that any number other than `0`, when converted to logical, is `TRUE`. However, when `"1"` is converted directly from character to logical, the result is `NA` – meaning "not available" or that it is not possible. However, if the `"1"` is first converted to a numeric, then converted to logical, the result is `TRUE`. Any comparison to an `NA` will be `NA`.

```
NA == 1
## [1] NA
```

Homework exercises

These exercises are to help you solidify and expand on the information given above. We intentionally added some concepts that were not covered above, and hope that you will take a few minutes to think through what is happening and how R is interpreting the code.

1. As you might expect the R operators can be combined in a single expression, e.g. `2 + 6/3`. Look up the order of operations within R and order the following operators from highest to lowest precedence:

   `-`, `>=`, `^`, `>=`, `<=`, `*`, `+`, `==`, `<-`, `/`

   Hint: some of the operators are considered equally. Try making a list where the top line has the operators with highest precedence, and any operators on the same line have the same precedence.

2. Like in algebra, parentheses can be used to specify the order of operations. What then would you expect to be the result of the following expressions, knowing the order of operations from exercise 1? (Try to predict the answer before typing the code into R.)

```
1 + 3*3
(1 + 3)*3
2^4/2 + 2
2^4/(2 + 2)
(5 + 2*10/(1 + 4))/3
```

3. Predict the vector and its class resulting from the following expressions:

```
c(1, 3, 5)
c("a", "b")
c(TRUE, TRUE, TRUE, FALSE)
c(1, TRUE, 10)
c("a", FALSE, 100, "dog")
c(as.numeric(TRUE), "fish", 2, "fish")
c(6, 7, as.numeric(FALSE), as.numeric("hello"))
as.logical(c(1, 0, 10, -100))
as.logical(c("TRUE", "false", "T", "F", "True", "red"))
as.numeric(as.logical(c(10, 5, 0, 1, 0, 100)))
```

4. Predict the result of the following expressions:

```
1 > 3
14 >= 2*7
"1" > "3"
as.logical(10) > 2
0 == FALSE
0 == as.character(FALSE)
0 == as.character(as.numeric(FALSE))
as.character(0) == 0
TRUE == 1^0
as.numeric(TRUE) == as.character(1^0)
```

```r
as.numeric("one") == 1

# These are some "bonus" concepts. How does R compare character values?
# Make some predictions, then run the code and see if you can figure out
# the rules for yourself. Then write your own expressions to test the rules!
"a" < "b"
"a" < "1"
"a2" > "a1"
"aaa" > "aa"
"a" > "A"
as.character(as.numeric(TRUE)) > FALSE
```